Local Applications

Front end utility

Thu, Jan 10, 2008

A local application (LA) is a tool that has been used in many ways to particularize a front end. It is a separately compiled and downloaded C function invoked by the underlying system code, thus logically becoming an extension to the system code. This note describes some of the features supported by such applications in a wide variety of "Classic" front ends.

Update task

The system code in a front end includes support for many common features. About a dozen tasks comprise this support. Chief among these tasks is the `Update` task, which is run every 15 Hz cycle in synchronization with the accelerator clock system. (Some installations are designed to operate at 10 Hz.) Although the `Update` task performs many duties each cycle, it can briefly be considered to access local hardware to update the local data pool, call all active local applications, and fulfill all active data requests that are due on that cycle.

LA operation

As stated above, each local application (LA) is written as a C function. It expects two arguments and returns nothing. The first argument is a pointer to a 24-byte structure in the entry of the nonvolatile Local Application Table (`LATBL`) entry. This structure is:

Field	*Size*	*Meaning*
`smPtr`	4	Context static memory block allocated during initialization call
`enable`	2	Enable Bit# operated to enable/disable the LA
`param[9]`	18	Additional parameters defined for use by the LA

The second argument of an LA is the reason for the call. The numeric values are:

Code	*Reason*
0	Initialization
1	Termination
2	(unused)
3	Cycle
4	Net
5	Serial

For most LAs, only reasons 0, 1, and 3 apply. The Net code is used for LAs that support a network protocol. The Serial code is used in special cases with serial port data.

The Initialization call is made when the Enable Bit transitions from"0" to "1". The LA is expected to allocate a static memory block for its own ongoing use while it is enabled, placing its address in the `smPtr` field of the above 24-byte structure. For every other call type, it is thereby reminded of the location of this allocated memory block.

The Termination call is made when the Enable Bit transitions from "1" to "0". The LA is expected to release any resources it has acquired during its operation, and to free its static memory block, placing a `NULL` in the `smPtr` field. Often, the Termination call is not actually used, as many LAs are enabled from the time a system is booted until forced to reboot again.

As long as the Enable Bit remains set to "1", the LA is called from the `Update` task with a Cycle call. The exact timing of this call depends upon where the "Call enabled LAs" entry (type `0x1D`) is placed in the `RDATA` table whose entries guide the update of the local data pool, but it is usually placed near the end, when the data pool is already filled with fresh data values from the locally connected hardware. The order of the calls to LA instances depends upon the order of the corresponding entries in the `LATBL`.

LA Uses

An LA can serve many uses. Historically, the original use was to support a closed loop or to respond to a trip of some hardware connected to that front end. On each call, the LA may access certain data readings from the local data pool, perform a calculation, and possibly update additional data pool readings. Consider a simple recent example. The local application `VACC` is designed to sample from the local data pool a voltage reading digitized from an ion gauge controller, calculate the corresponding exponent range and mantissa, and deliver the floating point result in units of Torr. Each exponent value represents a 1 volt change in the raw reading, and the mantissa in the range 1–9 amounts to a voltage fraction. The `ChanRaw` function is called to get the local voltage reading from the data pool, and the `SetFRead` function is called to install the floating point result into the local data pool. This same process is performed for each Cycle call, but more complex cases of an LA may need to average across multiple Cycle calls, or to implement suitable state machine logic as needed. All such details are the business of the LA; the system code does not care about it. The one assumption made by the system is that the LA will not use too much execution time during any single call, since the system code expects to comfortably keep up with the 15 Hz Cycle rate. In actual cases, front ends often have dozens of enabled LAs that are invoked by the `Update` task on every cycle.

The static memory block can also be used to provide various diagnostics of interest to the programmer. These are commonly accessed via the usual Memory Dump page, but they can also be accessed via a suitable data request. Another mechanism is to have the LA write records of particular interest to a data stream. As an example of the latter, the LA `AERS` runs in each front end node to shepherd delivery of alarm messages to the Acnet alarm handler called `AEOLUS`. For every transaction made with `AEOLUS`, it writes a 16-byte record to the `AERSLOG` data stream. When questions arise about Acnet alarm messages, this data stream can be useful to help resolve them. Again, whatever diagnostics make sense for the given LA are of no concern to the system code.

It is possible to have multiple instances of a given LA, so that the same executable code is called more than once each cycle, each time with a different set of parameters. A recent example is `PARC`, which regulates the preaccelerator arc current by tweaking the arc supply voltage. Since both preaccelerators connect to a single front end, it is natural to use two LA instances of `PARC`, one for each preaccelerator. The I/O signals are indicated by a separate set of parameters, including different Enable Bits, and a separate static memory block.

Memory file system

Install a given LA into a front end by writing the code into its local nonvolatile memory file system. For the case of 68040-based nodes, the files contain executable code. For PowerPC-based nodes, the files contain object code of the form loadable by VxWorks. When a program is initially started up, in any case, an executable image is copied into dynamic memory for subsequent calls by the `Update` task. All instances of the LA call the same image.

Downloading into a node's nonvolatile file system can be done in two ways. One is by using the TFTP protocol with the LA called `TFTP`, which serves as a TFTP protocol server, already installed in the node. This method is normally used to copy an initial or updated version of a file from a development node into a target front end. The other way is by using the Classic protocol to perform a sequence of settings to deliver the file to the target front end. (This is normally done via the Download page application, usually assigned to Page `D` in each front end.) One can even multicast an update to a given LA, so that all nodes addressed by the multicast address, and that currently have a previous version of the LA, can receive a new version of the file simultaneously. File versions are characterized by a version date, specified by Yr, Mo, Da, Hr, Mn. As the file is copied from one node to another, its version date is carried along. All LA file names have the form `LOOPxxxx`.

When a new version of a local application is downloaded to a node that already has that application enabled and running, the system automatically terminates the running code and initializes the new version. (In unusual cases for which this automatic switching logic is not appropriate, disable the old version first, so that the new version is loaded without being automatically enabled to execute.)

Page applications

A page application, as referenced above, is similar to a local application in that it is separately compiled and downloaded into the target node's memory file system. But it is brought into execution by the `Application` task, not the `Update` task, and only a single page application can be active at one time, whereas there can be many enabled local applications. Each node has a screen image that is the display of the currently active page application. The black and white screen image is alphanumeric (upper case only) and organized as 16 lines of 32 characters each. This design stems from that originally implemented in hardware driven by an inexpensive 6847 game chip *circa* 1980, with its video output driving a simple TV monitor. But even without the installed hardware, each front end supports page applications via what is locally termed Page `G`, named for the page to which it is normally attached on a front end's page application index. Page `G` client support exists for several platforms. Page applications are attached via index page support to pages identified by page numbers in base 32, denoted by `0`–`9`, `A`–`V`. Page `0` is always the index page. Other pages can be assigned to page application files by name, where the file name is of the form `PAGExxxx`. As examples, `PAGEPARM` is the front end Parameter page application; `PAGEDNLD` is the above-mentioned Download page application. Each page application is a C function that has two parameters. The first is a pointer to a 120-byte page-private nonvolatile memory area, which is for use in maintaining information to be preserved across separate callups of that page. (It can also allocate a static memory block, keeping a pointer to same in this 120-byte area.) The second parameter is similar to that used by local applications, with the following call reasons:

Code	*Reason*
0	Initialization
1	Termination
2	Keyboard interrupt (or Click)
3	Cycle

The term "Keyboard interrupt" is historic. It is a kind of "Do It" button, or nowadays, it corresponds to a mouse click. Its meaning normally depends upon the location of the cursor on the page display when the action is taken. The other three call reasons are the same as for local applications.

Access to LA params

When installing a local application instance, one normally uses the page application `LAPP`, which is normally assigned in a front end as Page `E`. One enters a node number and a `LATBL` entry number. The page display shows the list of the ten LA parameter values, including the Enable Bit number. Any can be modified via this user interface. A special test file is consulted to provide prompting text that identifies the meaning of each parameter, based upon the name of the LA assigned to that LATBL entry. The text file HELPLOOP is usually accessed from a library node when this page application is called up.

Data files

It is sometimes useful for an LA to read a data file that is separate from its own code. One use for this is to access a larger set of parameters than can be squeezed into 9 words. Within an LA, the code makes a local Classic request for the file contents. By convention, data files have names of the form DATAxxxx.

Utility programs

To assist in management of the nonvolatile files in various front ends, especially as compared with a "library" node that houses the latest versions of all files, use the page application VERS. Given a target node and a reference (library) node, it compares the files and lists any differences, marking which file is older/newer than the library version, or possibly not even contained within the library. It gets a copy of the CODES table (nonvolatile memory file directory) from each node and compares their contents.

Static memory access

To facilitate access to parts of the allocated static memory block for a given LA instance, there is a listype (96) that allows access to same. Its ident consists of a node number, a LATBL index and an offset. The index specifies the LA instance, and the offset reaches into the allocated static memory block in use by that LA instance. One can thus read or write it.

There is also a Data Access Table entry type (0x32) that can copy out words from a static memory block into the data pool. Again, an LATBL index is specified, along with an offset.

Obviously, one must take some care in accessing data in this way. If the structure definition of the static memory block changes, the offsets in use may no longer be valid. New fields should probably be added at the end.

There is a Print Memory utility page application (PMEM) that can print out parts of a static memory block for each instance of a specified LA. In place of the usual 8-digit memory address, enter a 4-character LA name appended by "+xxx", where xxx is the offset to use, specified in hex. This PA can list out the same information from an entire family of nodes, where a data file named DATANxxx includes the array of node numbers to be targeted. For example, the data file DATANLIN contains all the Linac front end node numbers.

Summary

Use of local applications in front ends has allowed a single relatively stable system code to be used for many projects, with additions made via local applications to satisfy particular needs of the project. Once a local application is installed, it acts as a permanent extension of the system code, being automatically initialized every time that front end boots.